Hochschule Kempten
University of Applied Sciences

EyeCandy3D User Manual

A 3D Scenegraph built on OpenGL

Inhaltsverzeichnis

1 Introduction

EyeCandy3D is a scene graph based on OpenGL. Every program consists of one **Application**, which holds a number of windows. Each window can hold a variable number of scenes, which can run simultaneously.

2 First steps

In this section I will guide you through the most important aspects of using this library.

2.1 Creating an application

An **Application** is the main part of this library.

```
Application app;
```

It allows the creation of new windows, which will automatically be visualized. The following function will create a new window and add it to the application:

```
app.createWindow<WindowType>(...);
```

WindowType has to be an own **Window** implementation.

2.2 Creating a window

Custom windows must be derived from **ec::Window**. This window has to be instanced with the window creation function provided by an **Application**.

```
class ExampleWindow : public ec::Window
{
public:
        explicit ExampleWindow(unsigned int windowWidth,
                unsigned int windowHeight,
                const std::string& title)
        // Own window implementation ...
};

ec::Application app;
app.createWindow<ExampleWindow>(width, height, "Example Window", "id
   ");
```

The dervied window **must** support a constructor with the following signature:
MyWindow(unsigned int, unsigned int, const std::string&)

2.3 Creating a scene

Custom scenes must be derived from **ec::Scene**. By registering it in the scene manager in the associated window, it can receive updates.

```
class ExampleScene : public ec::Scene
{
public:
        explicit ExampleScene(const std::string& name,
                ec::Window* window);

        // Own window implementation ...
};

// In ExampleWindow:
void ExampleWindow::initScenes()
{
        // Create new window
        auto exampleScene = new ExampleScene("example", this);

        // Register the new window, so it receives updates
        m_sceneSystem.registerScene(exampleScene);

        // ...
}
```

2.4 Resource registry

To ease the management of resources (images, geometry etc.), there is a **ResourceRegistry** class. Currently each window has a set of different resource registries, which can be used:

- ResourceRegistry< *Material* >
- ResourceRegistry< *Texture* >
- ResourceRegistry< *Geometry* >
- ResourceRegistry< *Drawable* >

Since these registries have no dependencies, further ones can easily be created. For shaders an extra class called **ShaderManager** is being used instead of a ResourceRegistry because shaders need extra treatment.

Scene graph 3

3.1 Nodes

The scene graph is made up of **Node**s. The graph is directed and has to be acyclic. Each scene contains one root node, which will always exist and cannot be deleted (but it can be transformed, i.e. translated). The root node doesn't have a parent.

Each node can have any number of children, but one father node at most. A node is a **Transform3D**, thus it can be translated, rotated and scaled. Besides a local matrix, it also contains a global matrix. Before each render cycle, its local and global matrices will be updated after the formula:

```
1: $\vec{x} := \vec{forward}$
2: $\vec{y} := \vec{up}$
3: $\vec{z} := \vec{x} \times \vec{y}$
4: $\vec{t} := \vec{pos}$
5: $M_{local} := (\vec{x}, \vec{y}, \vec{z}, \vec{t})$
6: $M_{global} := M_{\texttt{global\_parent}} * M_{local}$
```

3.2 Texture

The texutre class encapsulates a texture, which resides on the gpu. There are two functions, which allow the creation of either 2D or 3D textures.

```
class Texture
{
public:
        explicit Texture();
        ~Texture();

        /**
        * \brief Create a 2D texture.
        * \return True if creation was successful, false otherwise.
        */
        bool textureFromFile(const char* path, const std::string&
            type);

        /**
        * \brief Create a 3D texture.
        * \return True if creation was successful, false otherwise.
        */
        bool cubeMapFromFile(const char* path, const std::string&
           type);
};
```

3.3 Material

A **Material** contains a number of properties, which defines the color of an object. These properties include:

- Ambient, Diffuse, Specular colors
- Shininess factor for highlights
- Textures

```
class EC3D_DECLSPEC Material
{
        // ...

private:
        /* Flat colors */
        glm::vec4 m_colorAmbient;
        glm::vec4 m_colorDiffuse;
        glm::vec4 m_colorSpecular;
        glm::vec4 m_colorEmission;

        /* Additional attributes */
        float m_shininess;

        /* Textures */
        std::vector<Texture> m_textures;
}
```

There is also a convenience function for creating and adding a new texture to a material.

```
Material material;
material.addDiffuseTextureFromPath("Path/to/texture.png");
```

This is not recommended for larger projects, which often reuse textures.

3.4 Geometry

A geometry object encapsulates various geometry data, including:

- Vertices

- Normals
- Texture coordinates

It is responsible for memory managment on the GPU regarding geometry data.

There already are specific geometry implementations for common geometry types:

3D :

- CubeGeometry
- SphereGeometry
- CylinderGeometry

2D :

- CircleGeometry
- RectangleGeometry

If these predefined types aren't sufficicent, you can create your own geometry types. This is done by deriving from **StaticGeometry**.

3.5 Shader

The **Shader** class can load, compile and link shader programs. The shading language being used is GLSL, since OpenGL is being used.

3.6 Drawable

A drawable groups

- Geometry,
- Material and
- Shader

Drawables can be added to scene graph **Node**s so that they are being rendered.

Rendering 4

4.1 Renderer

Each window contains one **Renderer**, which is responsible for rendering to this window's back buffer. It contains one active **SceneRenderer**, which is responsible for the rendering process. The **SceneRenderer** can be changed, if a different combination of cameras should be used.

4.2 Scene Renderer

A **SceneRenderer** is responsible for rendering one or multiple scenes, by using a number of cameras, which the user defined to use.

4.3 Frame

A **Frame** contains a collection of cameras. It provides methods to add or remove cameras. The order of cameras matters, when viewports are overlapping. Cameras with lower index are prioritzed over cameras with a higher index.

4.4 Camera

Cameras are being used to define position and rotation of the view matrix, which is needed for rendering a **Scene**. Cameras are always linked to one specific **Scene**, so the **SceneRenderer** knows, which scenes to update and render.

Lighting 5

TBA

6 Input

Input is provided through the GLFW library.

6.1 Input events

Input events are always bound to one window. There are multiple sources, which can generate input events:

- **Mouse**
- **Keyboard**
- **Window**
- **Joystick**

An **InputEvent** consists of an **InputType** and the **EventData**. The **EventData** holds all different kinds of events, of which only one can be active at a time due to it being a union. The **InputType** describes, which part of the **EventData** is active. All other elements inside **EventData** are invalid!

I.e. if the **InputType** is *key_pressed*, only the KeyboardEvent inside the **EventData** is active.

6.2 EventSystem

An event system is always linked to exactly one window.

DeviceRegistry:
Contains input devices (mouse, keyboard, joystick etc.). Those devices can be activated to generate input events.

InputObservable:
InputListener can be registered at this component, which will then be notified about incoming **InputEvents**.

InputListener:
Input listener contain a number of callbacks, which can be added by the user. Input listener have to be registered at an **InputObservable** located in a window, to be informed about input events.

6.3 Device States

As an alternative of using the event system, one can directly access device states through the specific device class. An example for checking a specific keyboard state would be:

```
// Check if the left arrow key is being pressed
if(keyboard.isKeyDown(ec::Keyboard::LEFT))
{
        // Walk left...
}
```

This has the disadvantage that inputs can potentially be lost, if the user presses and releases a button before the program reaches this statement. It's like the button has never been pressed.

GUI 7

The GUI is built using the Agui-library with custom backends for OpenGL.

7.1 Initialization

Before the GUI library-part can be used, it has to be initialized. To do so, use the static function:

```
void ec::MiniAgui::init();
```

OpenGL and GLFW must be initialized before this function can be called! → Use it after creating an **Application** and atleast one **window**.

7.2 Backend

EyeCandy3D defines its own OpenGL backend for the Agui library, since Agui only provides backends for Allegro and SFML. The backend provides implementation for rendering shapes, images and fonts as well as for input.

7.3 Usage

Each **Camera** has one **GuiSystem**. You can retrieve this GuiSystem and provide it with your own implementation of a **agui::Gui**.